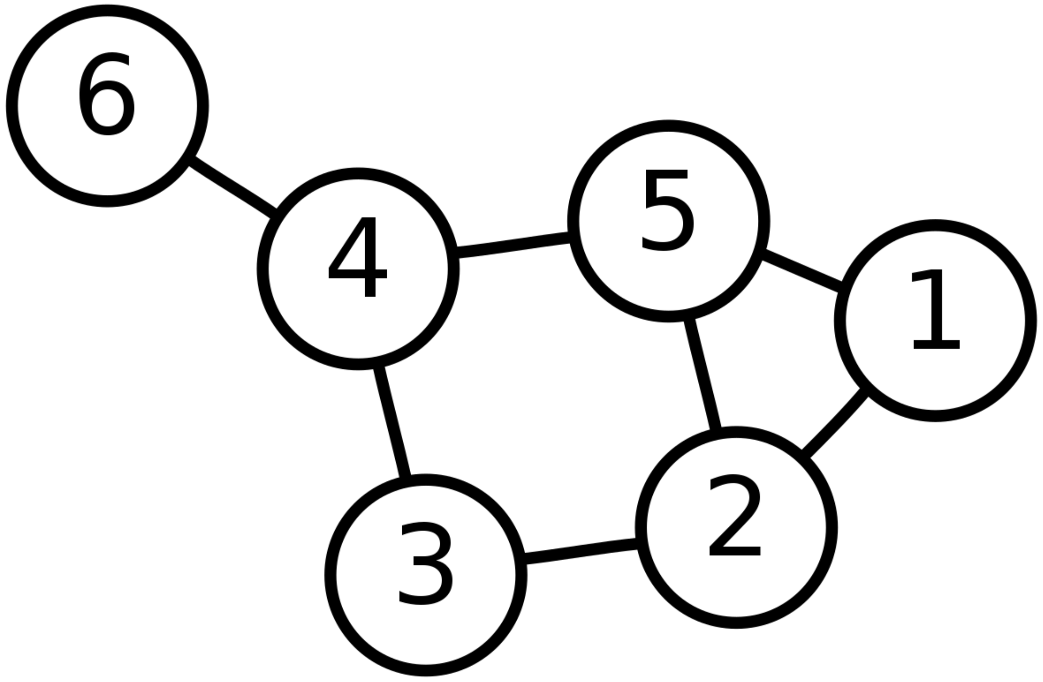
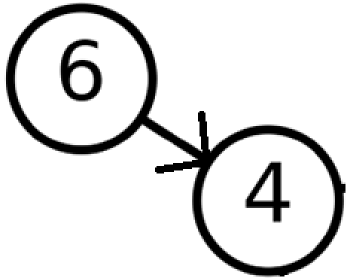
# Graphs

## Graphs

- A graph $G = (V, E)$ consists of a set of vertices $V$ and set of edges $E$
- Each edge is a pair $(v, w)$ where $v, w \in V$
    - If graph is undirected, the pair $(v, w)$ is unordered
- If pair $(v, w)$ is *ordered,* it is a *directed* graph
- Vertex $w$ is *adjacent* to $v$ if and only if $(v, w) \in E$
- In an undirected graph with edge $(v, w)$ and hence $(w, v)$, $w$ is adjacent to $v$ and $v$ is adjacent to $w$



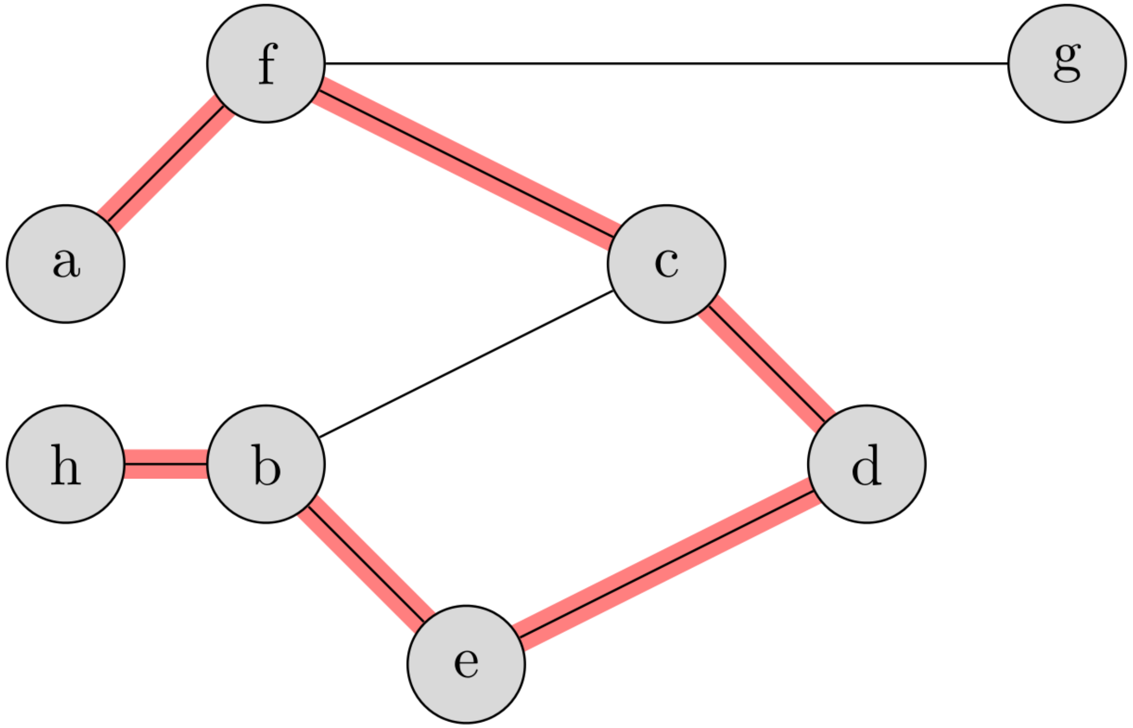
An **undirected graph**. $V = \{1,2,3,4,5,6\}$, $E = \{(2,1), (1,5), (2,3), (2,5), (3,4), (4,5), (4,6)\}$



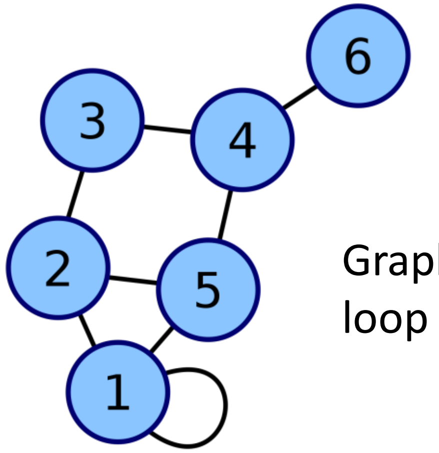
**Directed graph.** $V = \{6,4\}$ $E = \{(6,4)\}$

# Graphs

- A *path* in a graph is a sequence of vertices $w_1, w_2, w_3, \dots w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$.
- The *length* of such a path is the number of edges on the path, which is equal to $N - 1$
- We allow a path from vertex to itself, if it contains no edges, path length = 0
- If graph contains edge $(v, v)$ from a vertex to itself, the path $v, v$ is called a *loop*
- A *simple path* is a path such that all vertices are unique, except the first and last which can be the same



Undirected graph with path from $a$ to $h$ (or $h$ to $a$) highlighted in red
Path = $\{a, f, c, d, e, b, h\}$ or $\{h, b, e, d, c, f, a\}$
Length of path in both cases is 6, both paths are simple paths

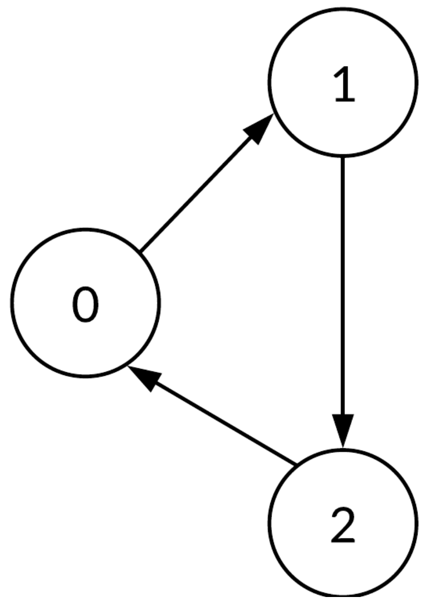


Graph containing a loop (1,1)

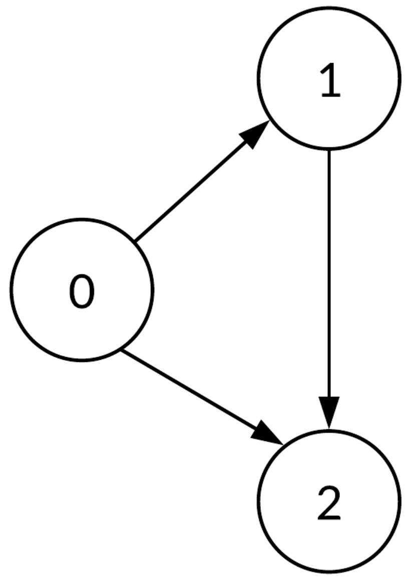
# Graphs

- A *cycle* in a directed graph is a path of length at least 1, such that $w_1 = w_N$
  - Cycle is simple if path is simple
  - For undirected graphs, require edges in a cycle be distinct
    - Path $u, v, u$ in undirected graph should not be considered a cycle, because $(u, v)$ and $(v, u)$ are the same edge
- Directed graph is *acyclic* if it has no cycles, sometimes referred to as a DAG (directed acyclic graph)
- Undirected graph is *connected* if there is a path from every vertex to every other vertex
  - Directed graph with this property is called *strongly connected*
- A directed graph is *weakly connected* if the underlying graph is connected when removing directions from edges
- A *complete* graph is a graph with a connection between every pair of vertices

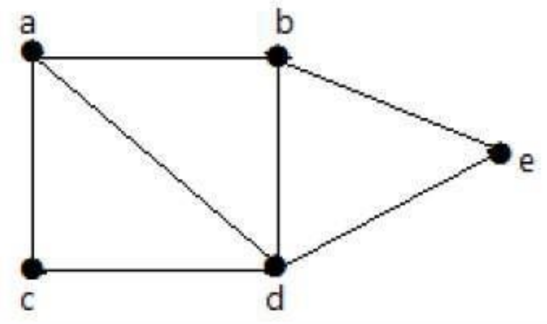
Cyclic Graph



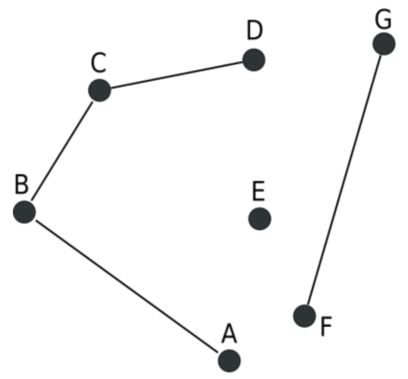
Acyclic Graph



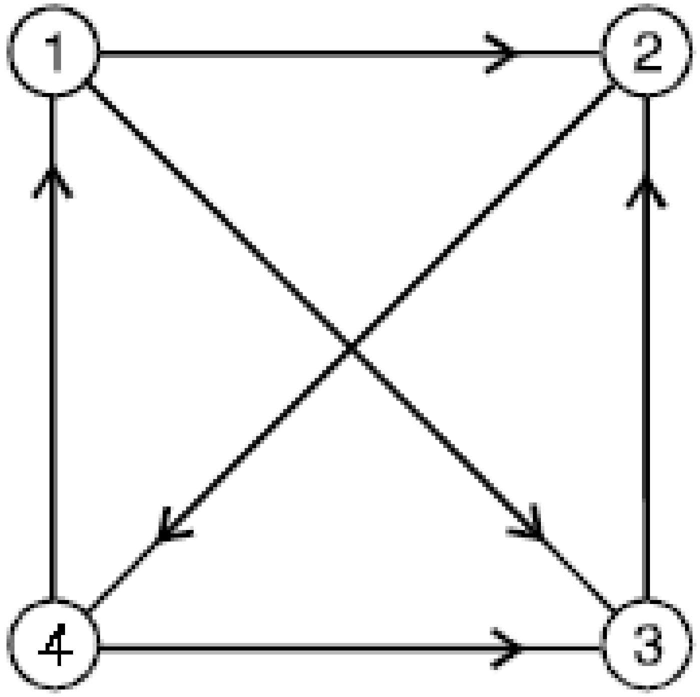
Connected graph (undirected)



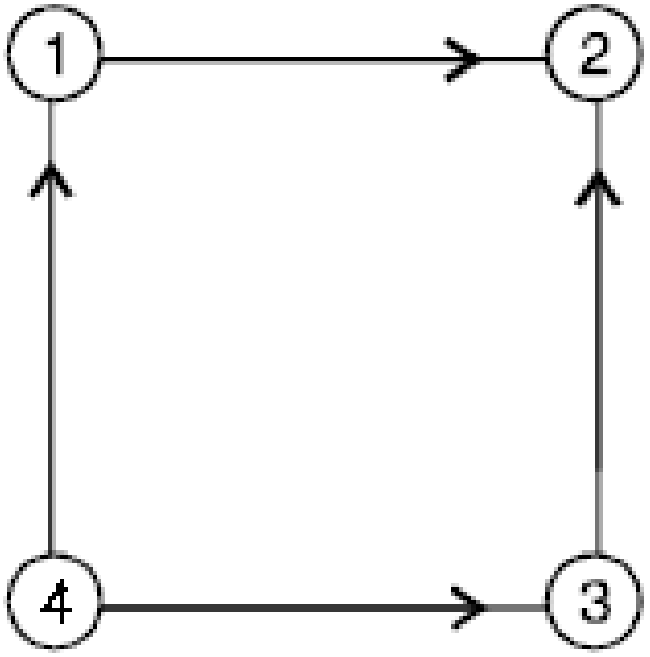
Disconnected graph (undirected)



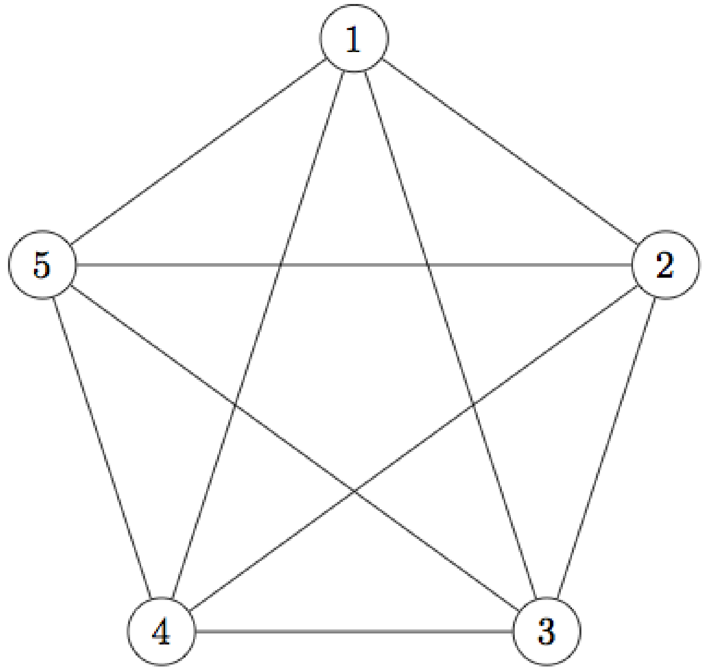
Strongly connected



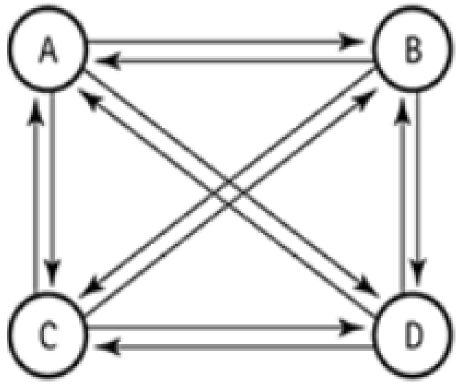
Weakly connected



Complete (undirected)



Complete (directed)



# Representation of graphs

- Figure 9.1: a directed graph with 7 vertices an 12 edges
- Can represent this graph using a 2-dimensional array or *adjacency matrix*
    - For each edge $(u, v)$ set $A[u][v]$ to true, otherwise, an entry in the array is false
    - If edge has weight associated to it, set $A[u][v]$ equal to the weight
    - Space complexity is $\Theta(|V|^2)$
    - Only appropriate if graph is *dense:* $$|E| = \Theta(|V|^2)$$
- If graph is *sparse,* can use *adjacency list* representation
    - For each vertex, keep a list of all adjacent vertices. Space requirement: $O(|E| + |V|)$
    - Can use a map (map vertices to lists) or keep each adjacency list as a data member of vertex class

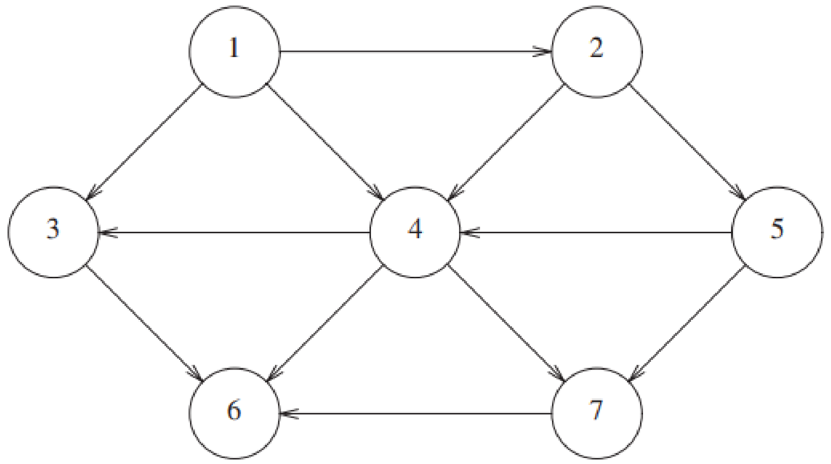


**Figure 9.1** A directed graph

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | | x | x | x | | | |
| 2 | | | | x | x | | |
| 3 | | | | | | x | |
| 4 | | | x | | | x | x |
| 5 | | | | x | | | x |
| 6 | | | | | | | |
| 7 | | | | | | x | |

Adjacency matrix representation of graph in 9.1

| | |
|---|---|
| 1 | 2, 4, 3 |
| 2 | 4, 5 |
| 3 | 6 |
| 4 | 6, 7, 3 |
| 5 | 4, 7 |
| 6 | (empty) |
| 7 | 6 |

**Figure 9.2** An adjacency list representation of a graph

# Graph algorithms: topological sort

- *Topological sort* is an ordering of vertices in a directed acyclic graph (DAG) such that if there is a path from $v_i$ to $v_j$, then $v_j$ appears after $v_i$ in the ordering
- *Example:* the course prerequisite structure at a University in Miami
  - Directed edge $(v, w)$ indicates that course $v$ must be completed before course $w$ may be attempted
- A topological ordering of these courses is any course sequence that does not violate the course prerequisite requirement

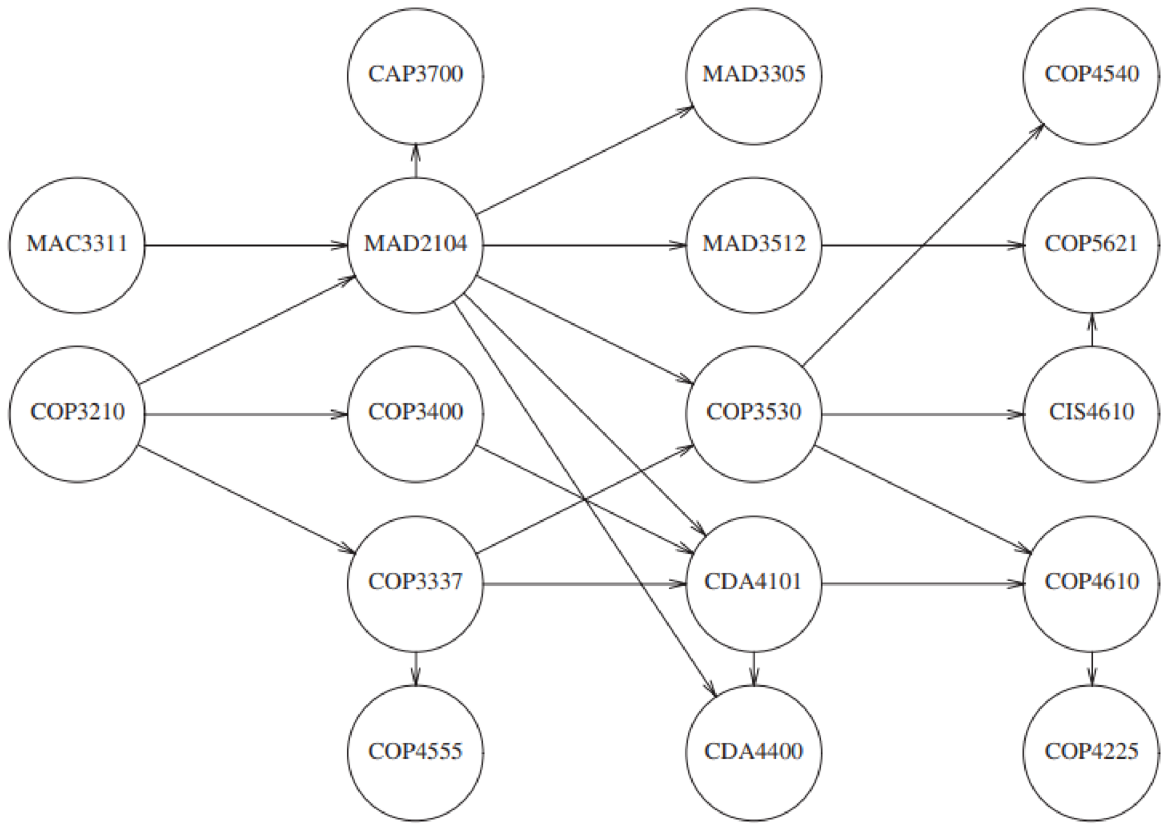


**Figure 9.3** An acyclic graph representing course prerequisite structure

## Topological sort

- Topological ordering not possible if graph has a cycle, since for any two vertices $v$ and $w$ in cycle, $v$ precedes $w$ and $w$ precedes $v$
- Ordering is not necessarily unique; any legal ordering will do:
  - $v_1, v_2, v_5, v_4, v_3, v_7, v_6$ and $v_1, v_2, v_5, v_4, v_7, v_3, v_6$ are both topological orderings of graph in figure 9.4

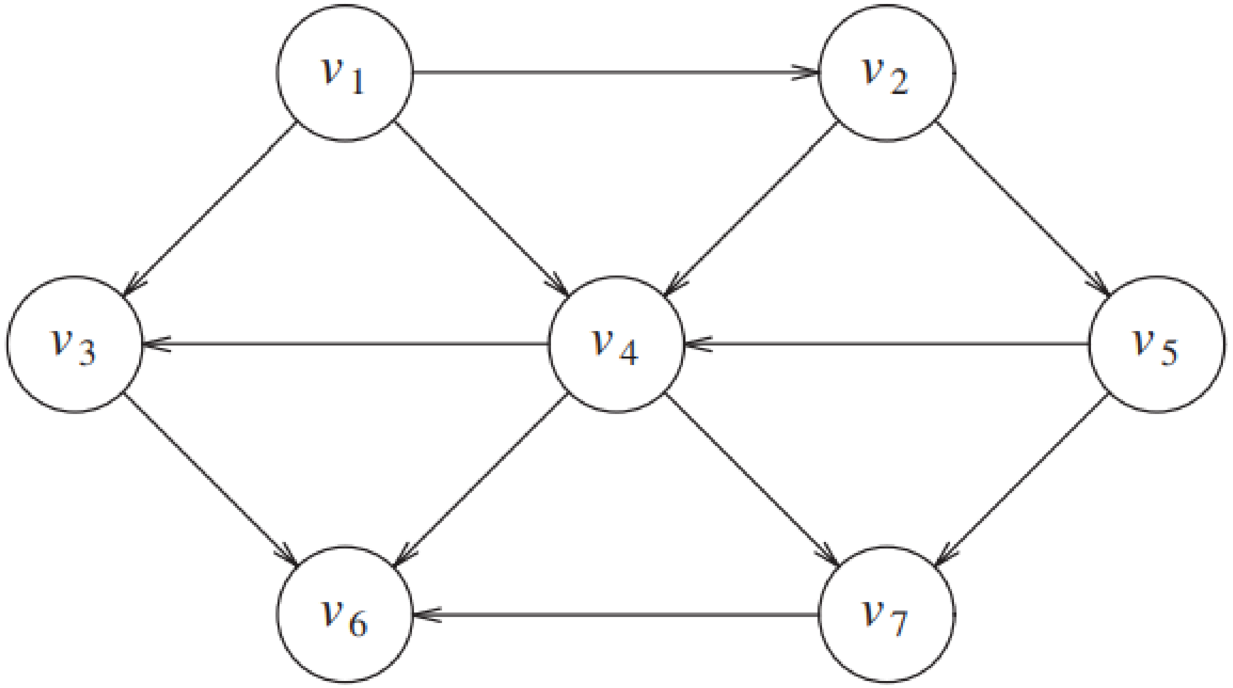


**Figure 9.4** An acyclic graph

# Topological sort

- Simple algorithm:
  - 1) find any vertex with no incoming edges, print it, and remove it along with its edges from the graph
  - 2) apply strategy in (1) to rest of graph repeatedly
- Formally:
  - The **indegree** of vertex $v$ is number of edges $(u, v)$
  - Store indegree for each vertex in graph, and store graph as adjacency list, can then apply algorithm in figure 9.5 to generate topological ordering
  - `findNewVertexOfIndegreeZero`: scans array of vertices looking for a vertex with indegree 0 that has not been assigned a topological number, returns NOT_A_VERTEX if no such vertex exists, indicating that graph has a cycle. Takes $O(|V|)$ time
  - $O(|V|)$ calls to findNewVertexOfIndegreeZero, so algorithm takes $O(|V|^2)$ time

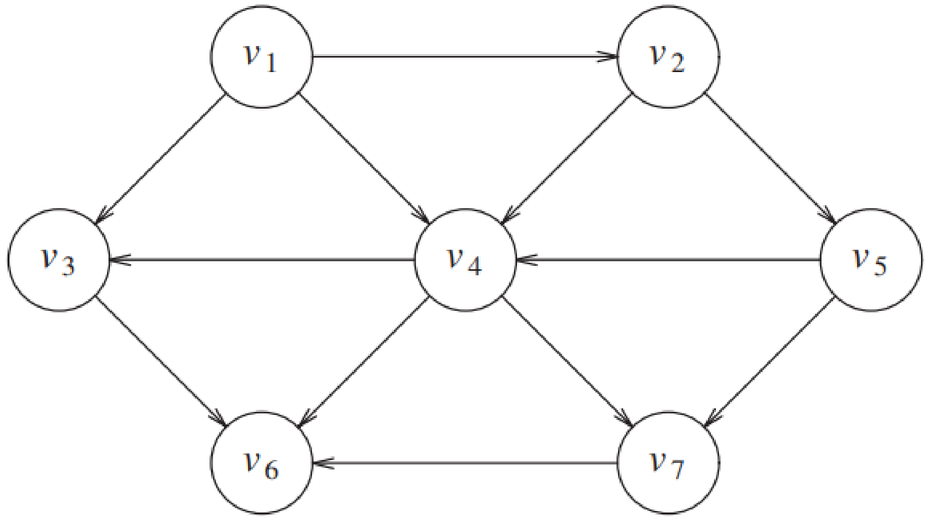
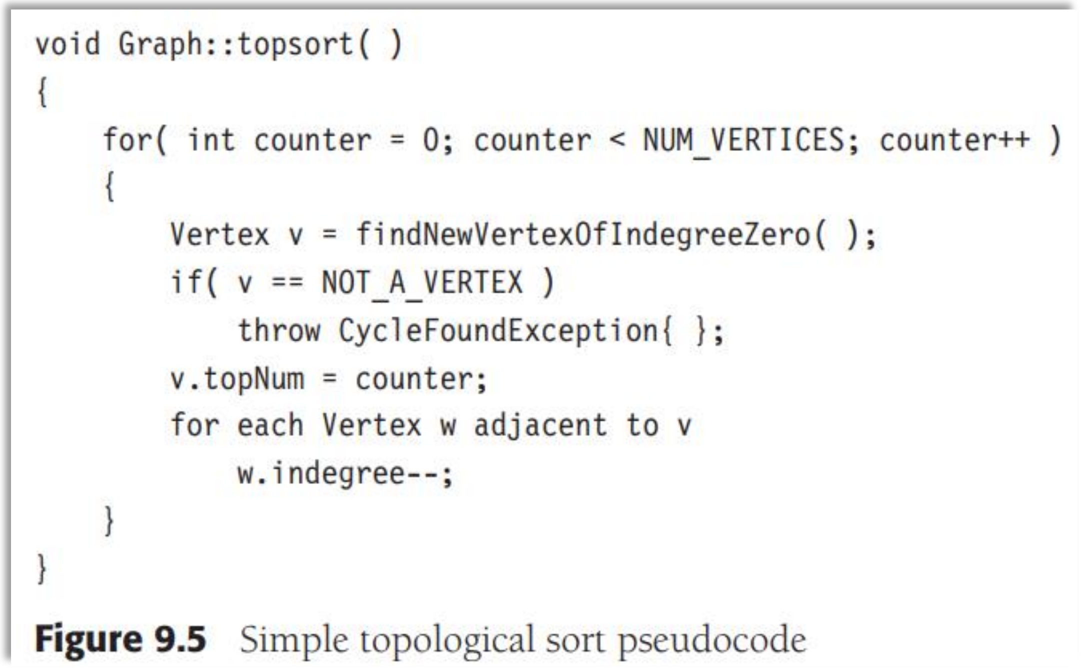


**Figure 9.4** An acyclic graph



```
void Graph::topsort( )
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == NOT_A_VERTEX )
            throw CycleFoundException{ };
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

**Figure 9.5** Simple topological sort pseudocode

# Example

| | Indegree Before Dequeue # | | | | | | |
|---|---|---|---|---|---|---|---|
| Vertex | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $v_1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $v_2$ | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $v_3$ | 2 | 1 | 1 | 1 | 0 | 0 | 0 |
| $v_4$ | 3 | 2 | 1 | 0 | 0 | 0 | 0 |
| $v_5$ | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| $v_6$ | 3 | 3 | 3 | 3 | 2 | 1 | 0 |
| $v_7$ | 2 | 2 | 2 | 1 | 0 | 0 | 0 |
| *Enqueue* | $v_1$ | $v_2$ | $v_5$ | $v_4$ | $v_3, v_7$ | | $v_6$ |
| *Dequeue* | $v_1$ | $v_2$ | $v_5$ | $v_4$ | $v_3$ | $v_7$ | $v_6$ |

**Figure 9.6** Result of applying topological sort to the graph in Figure 9.4

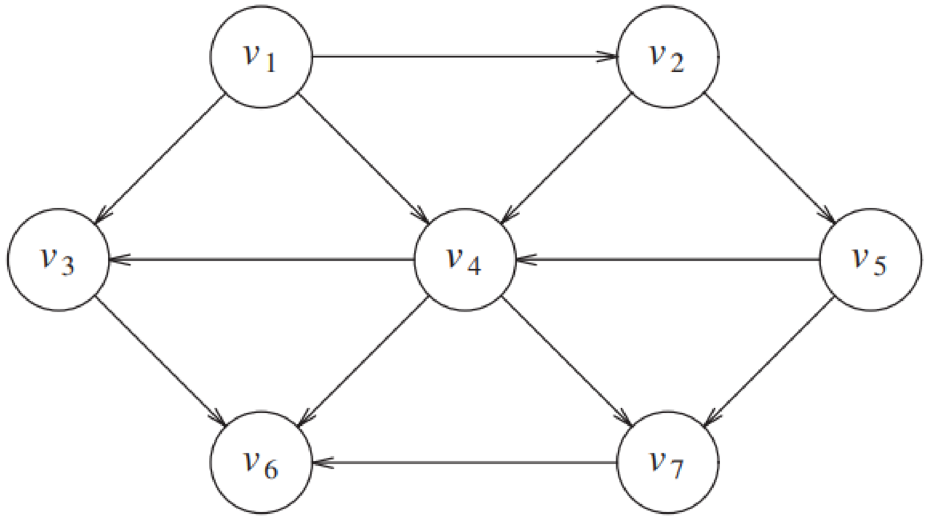
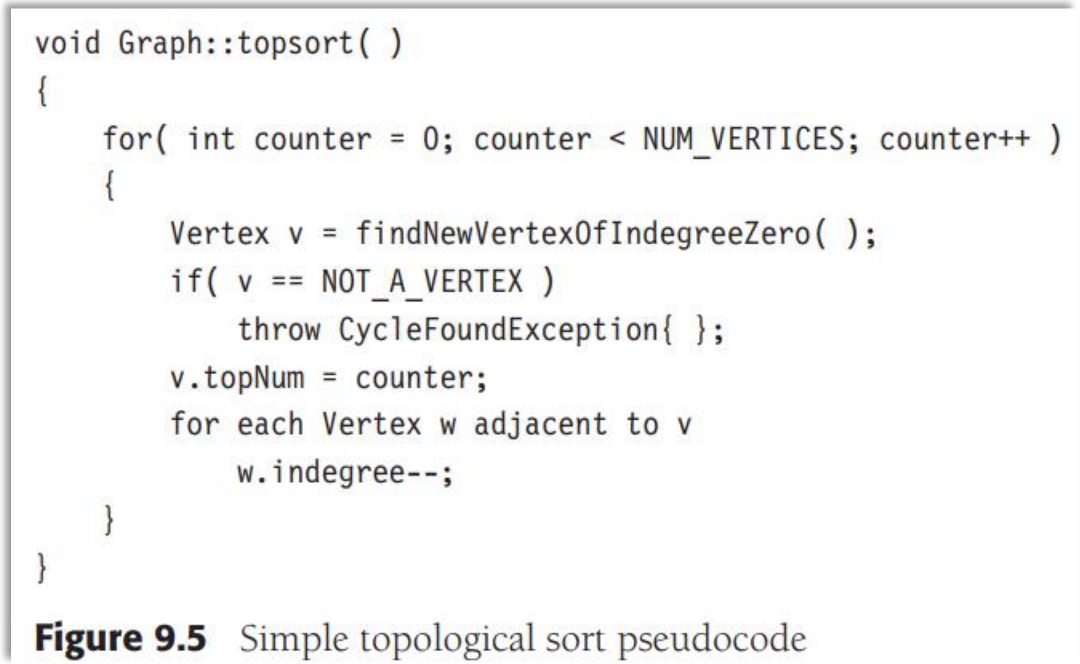


**Figure 9.4** An acyclic graph



```
void Graph::topsort( )
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == NOT_A_VERTEX )
            throw CycleFoundException{ };
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

**Figure 9.5** Simple topological sort pseudocode

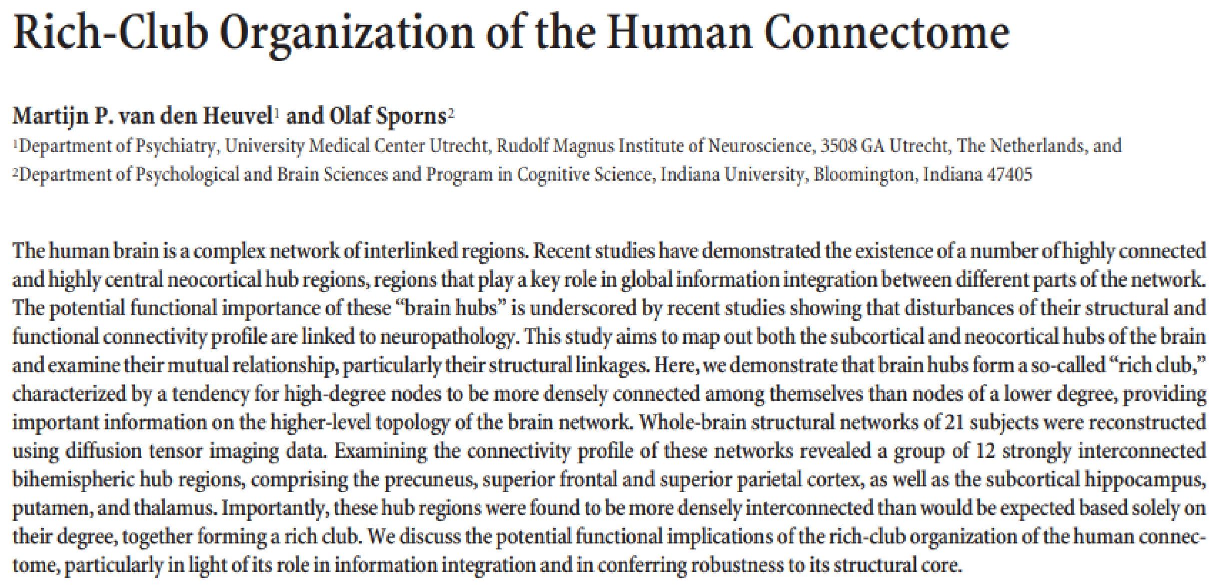
# Improving running time of topological sort

- Can do better than $O(|V|^2)$ by selecting more carefully our data structures
- Slow running time is caused by repeatedly performing a sequential scan through all vertices
    - If graph is sparse, it's likely that only a few vertices had their indegree updated after removing a vertex, but in the search for a new vertex of indegree 0, we look at potentially all vertices even though only a few have changed
- Can improve running time by using a queue to store vertices of indegree 0.
    - 1) compute indegree of all vertices (as before)
    - 2) all vertices of indegree 0 placed in empty queue
    - 3) while queue is not empty, vertex $v$ is removed, and all vertices adjacent to $v$ have their indegrees decremented
    - 4) vertex is placed on queue as soon as it's indegree falls to 0
    - Topological ordering is then the order in which vertices dequeue
- Time complexity is now $O(|E| + |V|)$ (if adjacency list used) instead of $O(|V|^2)$

```
void Graph::topsort( )
{
    Queue<Vertex> q;
    int counter = 0;

    q.makeEmpty( );
    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw CycleFoundException{ };
}
```

**Figure 9.7** Pseudocode to perform topological sort

# Graph applications

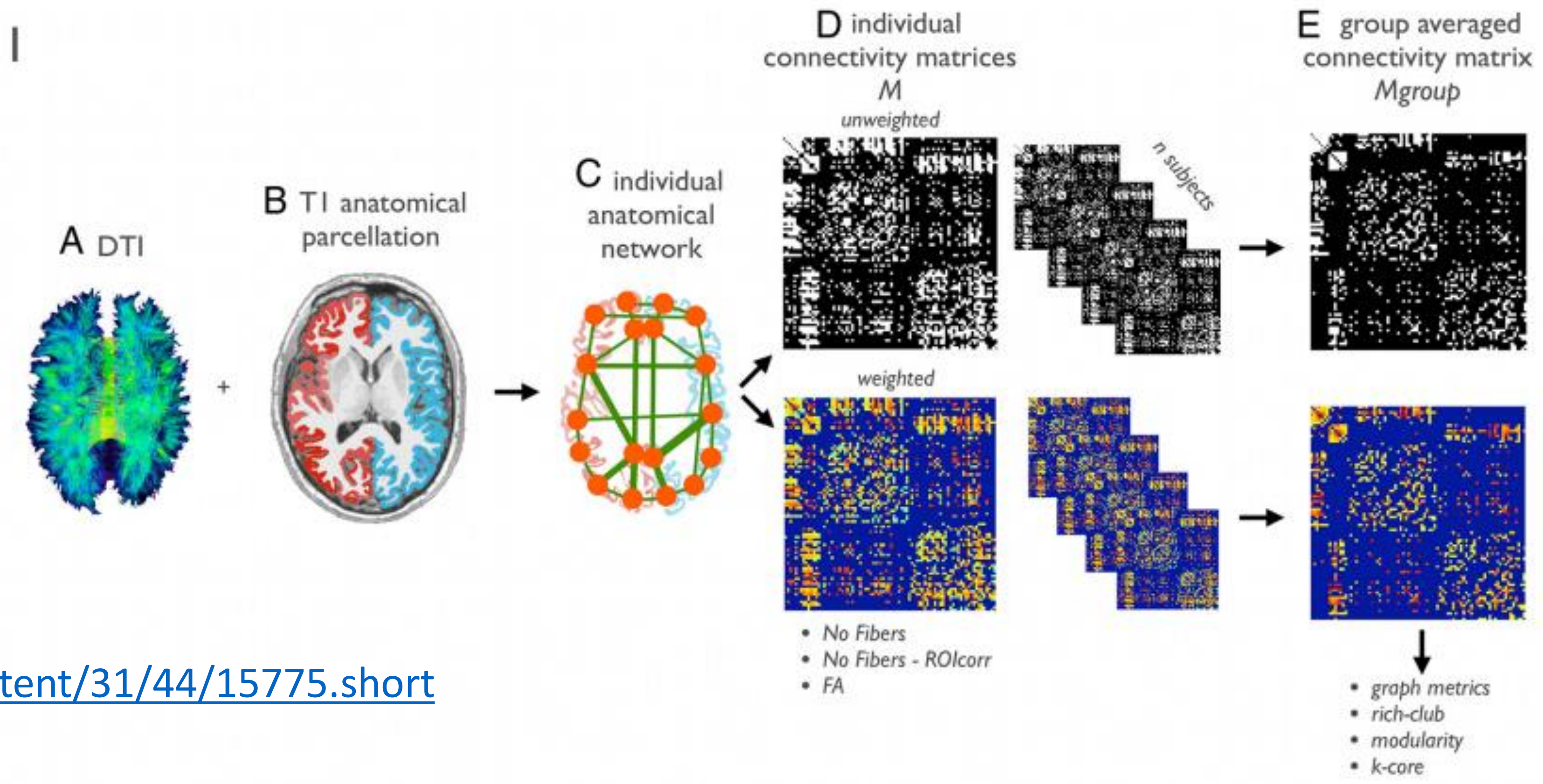


## Rich-Club Organization of the Human Connectome

**Martijn P. van den Heuvel**[1] **and Olaf Sporns**[2]

[1]Department of Psychiatry, University Medical Center Utrecht, Rudolf Magnus Institute of Neuroscience, 3508 GA Utrecht, The Netherlands, and
[2]Department of Psychological and Brain Sciences and Program in Cognitive Science, Indiana University, Bloomington, Indiana 47405

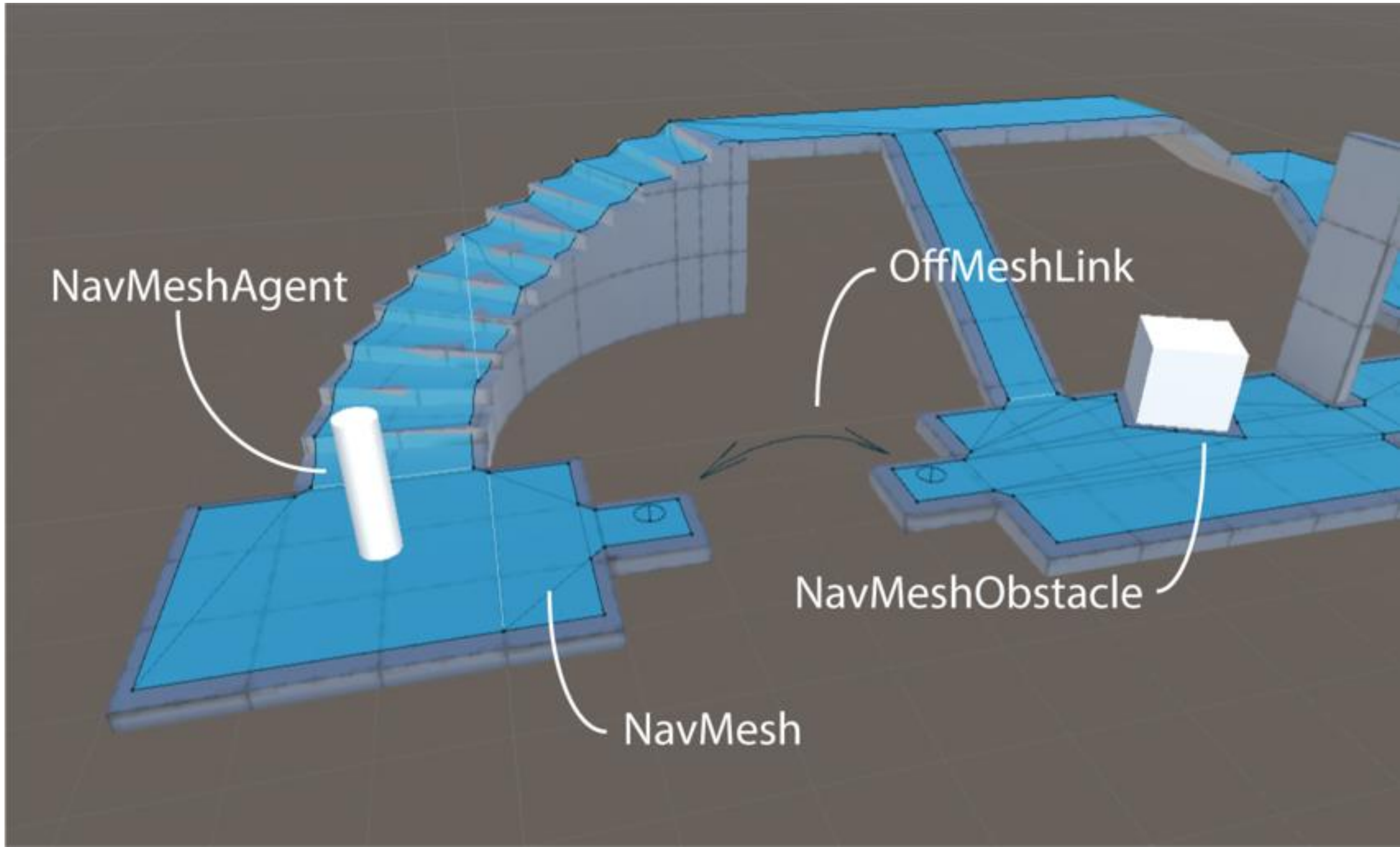
The human brain is a complex network of interlinked regions. Recent studies have demonstrated the existence of a number of highly connected and highly central neocortical hub regions, regions that play a key role in global information integration between different parts of the network. The potential functional importance of these "brain hubs" is underscored by recent studies showing that disturbances of their structural and functional connectivity profile are linked to neuropathology. This study aims to map out both the subcortical and neocortical hubs of the brain and examine their mutual relationship, particularly their structural linkages. Here, we demonstrate that brain hubs form a so-called "rich club," characterized by a tendency for high-degree nodes to be more densely connected among themselves than nodes of a lower degree, providing important information on the higher-level topology of the brain network. Whole-brain structural networks of 21 subjects were reconstructed using diffusion tensor imaging data. Examining the connectivity profile of these networks revealed a group of 12 strongly interconnected bihemispheric hub regions, comprising the precuneus, superior frontal and superior parietal cortex, as well as the subcortical hippocampus, putamen, and thalamus. Importantly, these hub regions were found to be more densely interconnected than would be expected based solely on their degree, together forming a rich club. We discuss the potential functional implications of the rich-club organization of the human connectome, particularly in light of its role in information integration and in conferring robustness to its structural core.


I
A DTI
B T1 anatomical parcellation
C individual anatomical network
D individual connectivity matrices M
unweighted
n subjects
weighted
• No Fibers
• No Fibers - ROIcorr
• FA
E group averaged connectivity matrix Mgroup
• graph metrics
• rich-club
• modularity
• k-core

https://www.jneurosci.org/content/31/44/15775.short

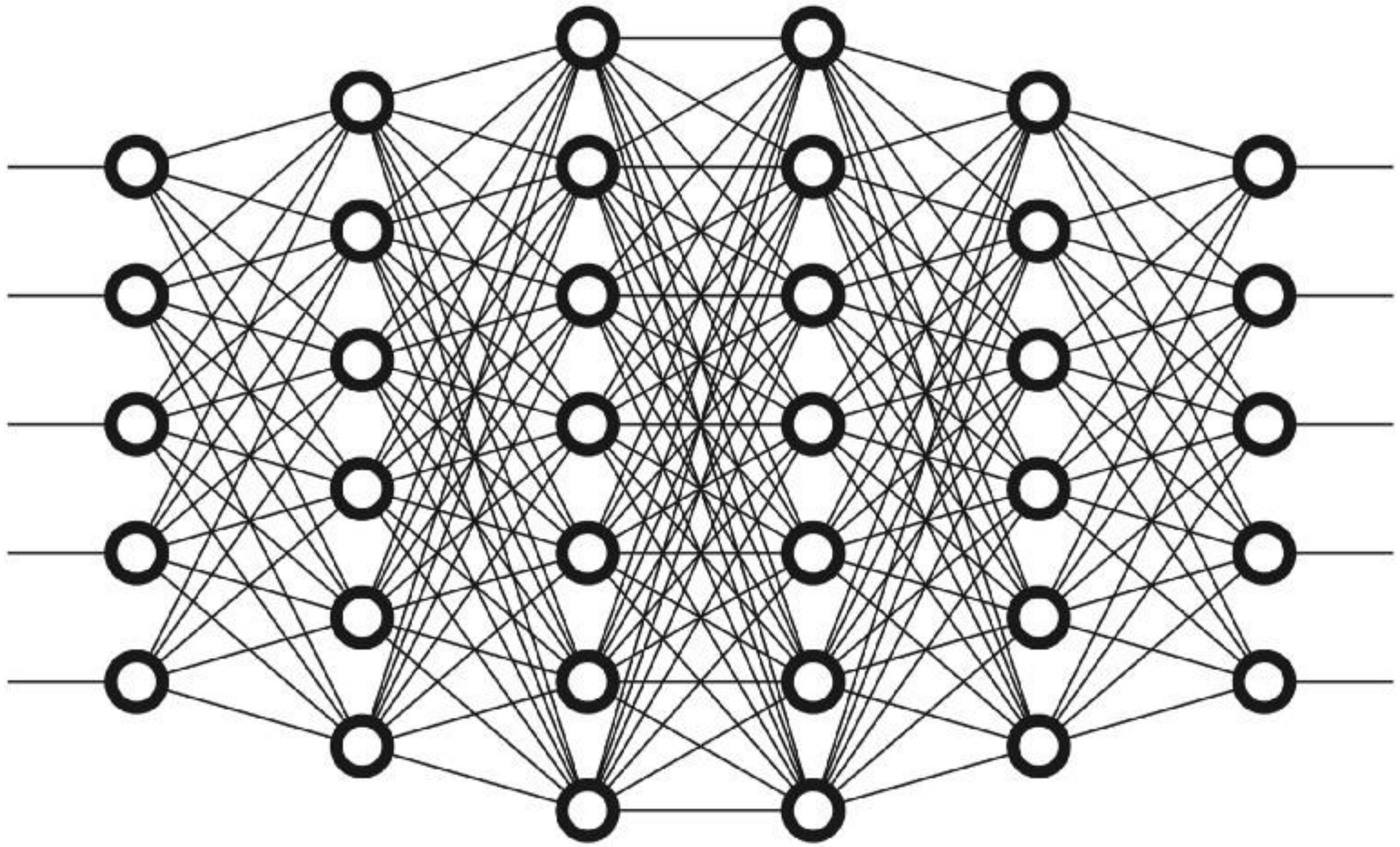
## Graph applications

https://docs.unity3d.com/Manual/nav-NavigationSystem.html



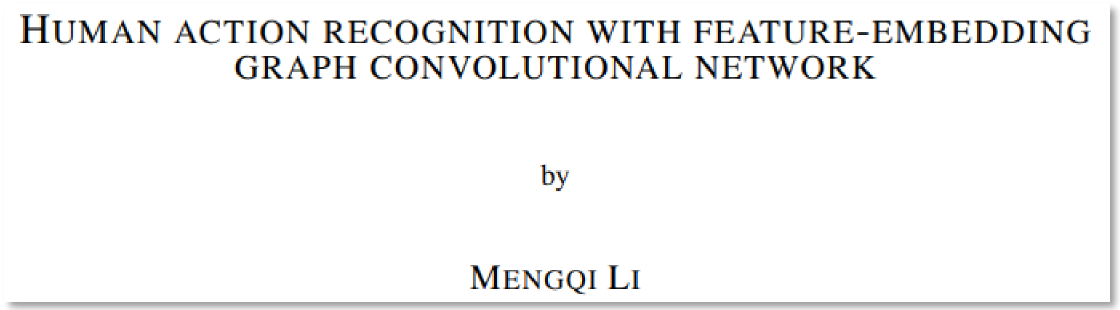
## Graph applications

https://en.wikipedia.org/wiki/Deep_learning



# Graph applications

https://www.ubishops.ca/wp-content/uploads/li20230106.pdf



HUMAN ACTION RECOGNITION WITH FEATURE-EMBEDDING GRAPH CONVOLUTIONAL NETWORK

by

MENGQI LI

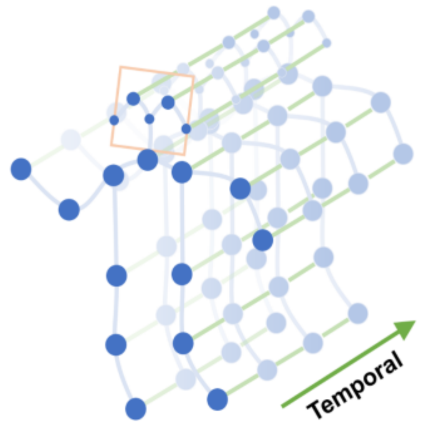


Figure 1.1: The spatial temporal graph of a skeleton sequence used in this work where the proposed ST-GCN operate on. Blue dots denote the body joints. The intra-body edges between body joints are defined based on the natural connections in human bodies. The inter-frame edges connect the same joints between consecu-tive frames. Joint coordinates are used as inputs to the ST-GCN, as well as our model.